Run-Time Management of Dynamically Reconfigurable Designs

N. Shirazi, W. Luk and P.Y.K. Cheung

Department of Computing, Imperial College, 180 Queen's Gate, London SW7 2BZ, UK

Abstract. A method for managing reconfigurable designs, which supports run-time configuration transformation, is proposed. This method involves structuring the reconfiguration manager into three components: a monitor, a loader, and a configuration store. Different trade-offs can be achieved in configuration time, optimality of the configured circuits, and the complexity of the reconfiguration manager, depending on the reconfiguration methods and the amount of run-time information available at compile time. The proposed techniques, implementable in hardware or software, are supported by our tools and can be applied to both partially and non-partially reconfigurable devices. We describe the combined and the partitioned reconfiguration methods, and use them to illustrate the techniques and the associated trade-offs.

1 Introduction

Exploiting the run-time configurability of FPGAs has been regarded by many as the key to overcoming their reduced capacity and speed compared with custom integrated circuit implementations. The approach will, however, only be valid if the time for reconfiguring the FPGAs does not outweigh its benefits of increasing capacity. Techniques are required to manage reconfigurable resources efficiently at run time; such techniques may also provide abstractions which hide low level details from users when appropriate.

This paper presents a method for efficient run-time management of reconfigurable designs, which involves structuring the reconfiguration manager into three components: a monitor, a loader, and a configuration store. The method can be implemented in hardware, software, or a combination of both. It can be applied to dynamically reconfigurable systems containing one or more FPGAs, which may or may not support partial reconfiguration. Techniques such as run-time transformation and partitioning the reconfiguration manager can be used to optimise configuration store usage or to reduce reconfiguration time.

Our work complements related research on tool development and run-time support for reconfigurable systems [1], [2], [3], [4], [8], [10]. The important aspects of our work include: (a) exploitation of compile-time information for optimising run-time performance, (b) flexibility of implementing the reconfiguration manager in hardware or software, (c) support for both partially reconfigurable and non-partially reconfigurable FPGAs.

2 Framework Overview

This section provides an overview of our framework for reconfiguration management. Details of the components in this framework will be presented later. While the discussion below centres on one dynamically reconfigurable FPGA, the framework can be extended to deal with multiple devices.

In this framework, the reconfiguration manager contains three components: a monitor, a loader, and a configuration store (Figure 1). The monitor maintains information about the configuration state, which may include the type and location of the circuits currently operating in the FPGAs. When the conditions for advancing to the next configuration state – such as receiving a request from the application or from the FPGA – are met, the monitor notifies the loader to install the new circuit at particular locations on the FPGA. In situations such as image processing, as long as the image size is fixed, the number of cycles for many operations are data independent and can be determined at compile time. The monitor can then be simplified to contain a few counters.

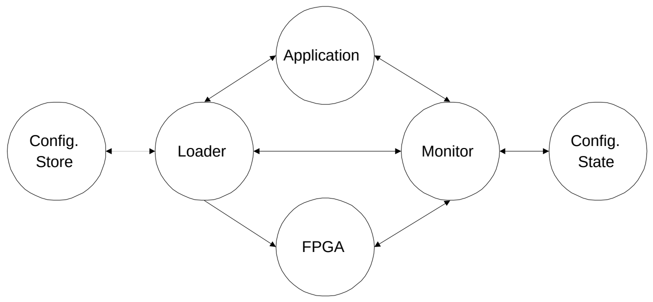

Fig. 1. Framework for reconfiguration manager.

The loader, on receiving a request from the monitor, configures the FPGA using data from a configuration store. When finished, it signals the monitor for completion, and normal operation can resume.

The configuration store contains a directory for the circuit configurations. The configurations are usually stored in the form of address-data pairs, where the data specify the configuration for an FPGA cell while the address indicates its location in the FPGA. A transformation agent can be used to transform or compose circuit configurations at run time; such details will be discussed later.

Our framework can be used to construct generic or customised reconfiguration managers. A generic reconfiguration manager can deal with a variety of applications, and is therefore likely to be more complex and less efficient. A customised reconfiguration manager is developed for one or a few applications, and can be optimised at compile time based on knowledge about run-time conditions. It is often more efficient, compact and simpler than a generic reconfiguration manager, but is not as flexible.

3 Design Flow

The proposed run-time management techniques are supported by a model [6] and the associated development procedure [7] for reconfigurable designs. There are six steps in this procedure: decomposition, sequencing, partial evaluation, incremental configuration calculation, simultaneous configuration generation, and validation. Reusable libraries [5], prototype tools [7], [9] and FPGA-based evaluation platforms [6] supporting these steps have been reported.

For this paper, we shall focus on the sequencing step. In this step, the design is captured as a network with control blocks connecting together the possible configurations for each reconfigurable component, together with the sequence of conditions for activating a particular configuration for each control block. In the next section, we shall describe how compile-time information captured in the activation sequence can be used to optimise the reconfiguration manager.

The above procedure can be explained using our model [6] for reconfigurable designs. In this model, a component that can be configured to behave either as A or as B is described by a network with A and B connected between two control blocks. The control blocks, RC_DMux and RC_Mux, route the data and results from the external ports x and y to either A or B at the desired instant, depending on the value c on their select lines (Figure 2). Each control block will be mapped either into a real multiplexer or demultiplexer to form a single-cycle reconfigurable design, or into virtual ones which model the control mechanisms for replacing one configuration by another [6]. We shall see how this model can be used in developing and optimising the reconfiguration manager in later sections.

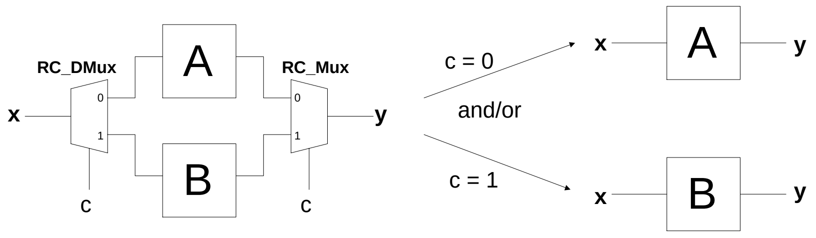

Fig. 2. A static network modelling a design that can behave either as A or as B, depending on the select value c to the control blocks RC_DMux and RC_Mux.

4 Monitor

The purpose of the monitor is to keep track of the configurations in the FPGA. The monitor also contains information about possible transitions to the next state from a particular state.

Since run-time conditions usually require rapid capture and may involve a large amount of data, part of the monitor often resides on the dynamically reconfigurable FPGA, and is mainly used for data-driven reconfiguration. The monitor checks for the user condition that activates reconfiguration. If the user condition for the next configuration is met and the desired configuration is not in a usable form on the FPGA, the monitor notifies the loader to introduce the configuration. When finished, the monitor may signal the completion of the configuration process if required.

The monitor includes one or more reconfiguration state machines. These state machines can be produced from our tools automatically and are based on the activation sequence from the user specifying the reconfiguration conditions (Section 3). A reconfiguration state machine indicates which configuration to load from the configuration store.

There are three possibilities for the monitor operation depending on the information in the reconfiguration sequence available at compile time.

(a) The duration for which the current configuration remains valid is known at compile time, and the next configuration is also known.
(b) The duration for which the current configuration remains valid is not known, although the next configuration is known.
(c) Both the duration for which the current configuration remains valid, and the next configuration, is not known.

Case (a) is the simplest: a timing mechanism such as a counter could be included in the monitor to indicate when the next configuration will be loaded. This happens, for instance, in video processing when the hardware reconfigures to a known next state after a fixed number of frames whose size is also known. Recall that RC_Mux/RC_DMux pairs are used to indicate the reconfigurable regions, and that changing the value on their select lines corresponds to reconfiguring between components delimited by the RC_Mux/RC_DMux pair (Figure 2). For case (a), these select lines will be connected to the timing mechanism.

For FPGAs supporting partial reconfiguration such as Xilinx 6200 devices, this means that partial reconfiguration will be performed after a fixed duration; for non-partially reconfigurable FPGAs such as Xilinx 4000 devices, entire chip configurations will be swapped. Provided that there is enough FPGA resources, one can implement the RC_Mux/RC_DMux pairs and the associated configurations as physical components on the FPGA to produce a single-cycle reconfigurable design [6], [9].

Case (b) requires inputs from run-time conditions, from the FPGA or from application software, to decide when the next configuration is required. In this case, the select lines of the RC_Mux/RC_DMux pairs are connected to the source that triggers reconfiguration. The same is true for case (c); however, since the choice of the next configuration is determined at run time, all possible next configurations will have to be produced at compile time or at run time.

Our scheme allows an abstraction layer above the RC_Mux/RC_DMux level. A mapping function can be defined that relates a value from the user design to the corresponding RC_Mux/RC_DMux pairs. In the constant adder example provided in Section 8, a user only needs to supply an integer constant which is then mapped to selecting the corresponding RC_Mux/RC_DMux pairs that indicate the reconfiguration to be performed.

Sometimes the designer can determine whether reducing the reconfiguration time, or optimising the size or speed of the new circuit, should take priority. For instance, one configuration may contain circuit elements usable by its successor, but in an suboptimal way. One can then decide whether to reduce the reconfiguration time and tolerate a suboptimal circuit, or to have a longer reconfiguration time in return for a better circuit. Alternatively, circuit elements from the next configuration can be included in the current configuration, such that circuit behaviour is preserved while reducing reconfiguration time. Facilities for estimating reconfiguration time will be useful [8].

5 Loader

The purpose of the loader is to carry out the reconfiguration of the FPGA, as specified by the select value for the RC_Mux/RC_DMux components. On receiving a request from the monitor, the loader obtains the location of the requested configuration from the configuration directory, extracts the configuration from the configuration store and then initiates the configuration process. On completion, the loader may, when appropriate, set a new clock speed for the new circuit. It then signals the monitor for completion, and normal operation can resume.

The software version of the loader runs on the host processor. API functions are provided to facilitate design development by hiding the mechanisms used for performing run-time reconfiguration. We follow an object-oriented approach, treating an RC_Mux/RC_DMux pair as objects which load a new configuration when the value on their select lines change. When the object is created, the configuration data associated with the RC_Mux/RC_DMux pair are loaded into the host's main memory to ensure fast configuration of the FPGA. The resulting facilities are similar to those supported by JERC [4].

To improve reconfiguration speed, we have developed a scheme to implement the loader in hardware. This enables dynamic reconfiguration to be performed at the maximum speed that the FPGA can handle. This is difficult to achieve by loading configurations from a loosely-coupled processor, for example an FPGA co-processor board that resides on a PCI bus.

A handshaking scheme is used to synchronise the user design with the reconfiguration manager, since the reconfiguration manager can be clocked faster than the user design. This allows multiple configuration cycles to occur in a single compute cycle, thus reducing reconfiguration overhead.

6 Configuration Store and Run-Time Transformations

The configuration store contains three components: a configuration directory, a repository for configuration data, and a transformation agent (Figure 3). The configuration directory and the configuration data can be arranged as shown in Figure 4. If required, the transformation agent transforms a configuration before loading it into the FPGA; this can be used in minimising configuration store usage, as discussed below. For performance critical applications, the transformation agent can itself be implemented in hardware. If the next configuration can be predicted at compile time or at run time before it is required, there may be sufficient time for a software transformation agent to perform its tasks.

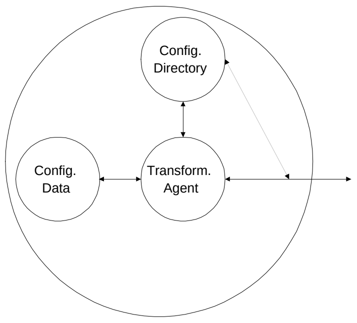

Fig. 3. Configuration store architecture. The configuration store is connected to the loader as shown in Figure 1.

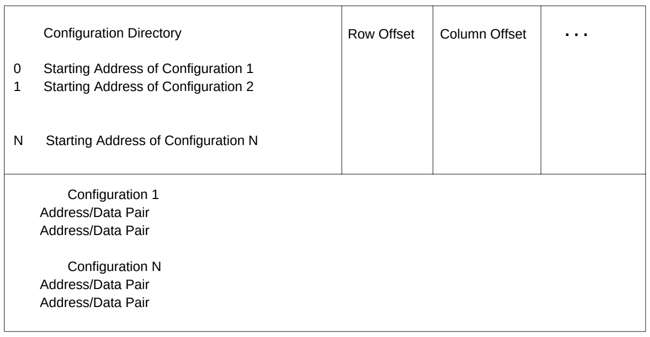

Fig. 4. Possible data arrangement in the configuration store, showing the configuration directory (top left), transformation parameters (top right) and configuration data (bottom). Row Offset and Column Offset are examples of transformation parameters for the configuration data which can be produced at compile time or at run time.

Fast storage is often scarce. To minimise configuration storage, three transformation methods are explored. The first method covers regular circuits: if the same configuration information is used in two or more locations of the FPGA, an offset (Figure 4) can be added repeatedly to the address of the base configuration to produce the required configurations. Our tools automatically calculate these offsets and the number of replications, and place them as transformation parameters in the configuration store. The replication of configuration data at the row and column offsets are generated by the transformation agent during reconfiguration of the FPGA.

The second method is to maximise sharing of lower-level components in the design hierarchy: for instance the same adder configuration can be used in producing different kinds of multipliers. This method is an extension of the first method to support hierarchical representations of configuration data.

The third method adopts a small number of configuration templates, which can be transformed by operations such as stretching or partial evaluation, for building the actual configuration bitstreams at run time. This method is particularly useful in, for example, producing constant-coefficient adders or multipliers. Further parameters can be included to support specific transformations.

All three transformation methods assume that the configurations are relocatable [10], and work best if there are minimum constraints on the placement of the circuits. These methods can be implemented in hardware to reduce their run-time overhead. While other configuration store architectures may result in greater utilisation, they may do so at the expense of increasing reconfiguration time or complicating the transformation agent.

7 Reconfiguration Methods

This section presents two reconfiguration methods, and assesses their impact on our framework. An example will be considered in Section 8; further case studies, such as arithmetic and video processing designs, are under development.

Combined reconfiguration method. For a design with n configurations, there are $n(n-1)$ possibilities of changing from one configuration to another. If the reconfiguration sequence is known at compile time, then we can generate incremental configurations instead of full configurations [7]. At run time, the transformation agent produces the required configuration from incremental configurations, including the computation of offsets (Section 6). For devices supporting partial reconfiguration or simultaneous reconfiguration, there will be an improvement in reconfiguration time since only the parts that change need to be reconfigured.

However, if the reconfiguration sequence is only available at run time, then up to $n(n-1)$ configurations will need to be generated at compile time. Alternatively the configurations will have to be produced on demand at run time.

Partitioned reconfiguration method. An alternative method is based on the principle that more efficient implementations can often be obtained by moving the RC_Muxes and RC_DMuxes to a lower level of description [6]. For the above example, this method is applicable if the n configurations can each be decomposed into m components, so that each component is controlled by its group of RC_Mux/RC_DMux pairs. m reconfiguration state machines are generated, one for each group of RC_Mux/RC_DMux pairs, so that the design can be configured to be one of the n possible configurations.

At run time, the required configuration is produced by the transformation agent from data for each of the m components. The configuration state machine in the monitor for each component determines if the conditions for transition has been reached; if so, it signals the loader to load the appropriate partial configuration.

In this example, the partitioned reconfiguration method reduces the number of partial reconfigurations from $n(n-1)$ to an application-specific value dependent on m. However, the reconfiguration controller is more complex than that for the combined reconfiguration method, since there are now m reconfiguration state machines instead of one. This method may not be able to take advantage of simultaneous reconfiguration techniques, unless the relevant control information (such as wildcard data for the Xilinx 6200 FPGA) can be computed rapidly [7]. Finally, a mapping function may be required to produce the appropriate control information for the m state machines; this will be illustrated in the next section.

8 Constant Adder

In this example, a bitslice of a variable adder is partially evaluated, resulting in the two circuits shown in Figure 5(a) which correspond to a constant zero adder and a constant one adder. Our tools [9] automatically find the reconfigurable regions in these two designs and insert RC_Muxes and RC_DMuxes to delimit the reconfigurable regions, resulting in the bitslice in Figure 5(b). This bitslice can then be replicated to give a constant adder of a particular size. For a Xilinx 6200 FPGA, the use of a constant adder in place of a variable adder reduces the size by 50%, and increases the speed by 33%.

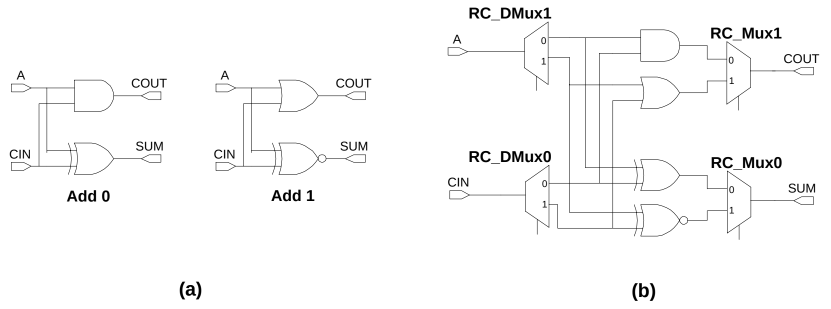

Fig. 5. (a) Two circuit bitslices for adding a constant zero or a constant one. (b) A circuit that can be reconfigured to implement either of the circuits in (a), by a select value at the control inputs of the control components RC_Mux and RC_DMux. The same select value is used for all four control components.

Combined reconfiguration method. In this method, the user specifies the constants in the command file along with the duration between reconfiguration if available. The configuration state diagram in Figure 6(a) is produced by our tools. If the duration between reconfiguration is known at compile time, then a timing mechanism will be included in the monitor to trigger the reconfiguration automatically. If the duration is not known, then the monitor keeps track of the configuration state so that, when the conditions for reconfiguration occur, it requests the loader to initiate the reconfiguration.

For this method, the ease of reconfiguration comes at the expense of increasing the amount of configuration data. For each bit that differs between two successive constants, two configurations cycles are needed in the Xilinx 6200: one for reconfiguring the XNOR gate to the XOR gate, and the other for reconfiguring the OR gate to the AND gate. Our tools can take advantage of device-specific optimisation such as wildcarding in the Xilinx 6200, thus reducing the amount of reconfiguration cycles between the constant "1111" and "0000" [7]. There are a total of 20 configuration words for the reconfiguration sequence in Figure 6(a). In general, if the user would like to reconfigure between all 2^n different constants, $2^n(2^n-1)$ partial configurations would have to be generated and stored.

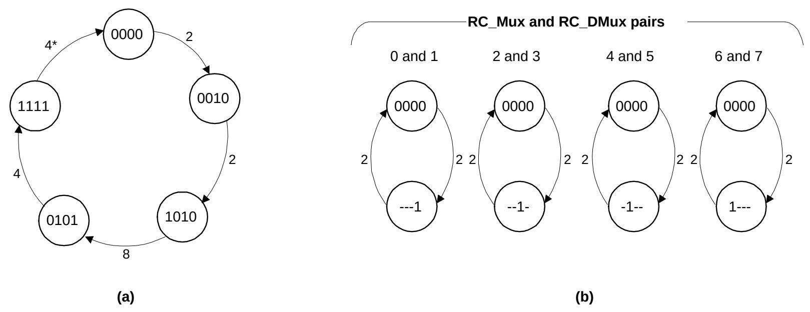

Fig. 6. (a) State diagram for incremental configuration of a 4-bit constant adder using the combined reconfiguration method. The number of configuration words involved in a transition is shown next to the corresponding edge. The asterisk indicates that the number of configuration words has been reduced by wildcarding. (b) State diagram for configuring each bitslice individually. RC_Mux/RC_DMux pairs correspond to the ones in Figure 5(b). A dash indicates a don't care condition for a particular bit.

Partitioned reconfiguration method. An alternative is to partition the adder into bitslices, and calculate the configuration needed for each bitslice to add a 0 or 1. To change a constant, a mapping function is defined that selects the appropriate RC_Muxes/RC_DMuxes for each bitslice shown in Figure 5(b). The monitor has access to a reconfiguration state machine for each bitslice, which determines if its bit of the constant has changed; if so, it signals the loader to load the appropriate partial configuration. The configuration for each bitslice is stitched together by the transformation agent to form the required configuration.

This method significantly reduces the amount of configuration data for an n-bit constant adder. Four configuration words are needed for each bitslice. Apart from the component at the least significant bit position due to the external carry input, the configuration bits for the bitslices are the same, except for an address offset. Hence we only need to store the configuration bits for the component at the least significant bit position and the repeating bitslice. During reconfiguration, the transformation agent in the configuration store adds the corresponding offsets to reconfigure the bitslice. There are only 8 configuration words needed to be stored using this method.

9 Summary

This paper presents a framework for efficient run-time management of reconfigurable designs, which exploits compile-time information for optimising run-time performance. The reconfiguration manager can be implemented in hardware or software, and supports both partially and non-partially reconfigurable FPGAs. Current and future work includes refining and extending our framework and tools, exploring their use in multi-tasking systems, and applying them to realistic applications.

Acknowledgements

The support of Xilinx, Interval Research, Hewlett Packard Laboratories Bristol and the UK Engineering and Physical Sciences Research Council (Grant Number GR/L24366, GR/L54356 and GR/L59658) is gratefully acknowledged.

References

1. G. Brebner, "A virtual hardware operating system for the Xilinx 6200", in *Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, LNCS 1142, Springer, 1996.
2. J. Burns et. al., "A dynamic reconfiguration run-time system", in *Proc. FCCM97*, IEEE Computer Society Press, 1997.
3. S. Cadambi et. al., "Managing pipeline-reconfigurable FPGAs", in *Proc. FPGA98*, ACM Press, 1998.
4. E. Lechner and S.A. Guccione, "The Java environment for reconfigurable computing", in *Field Programmable Logic and Applications*, LNCS 1304, Springer, 1997.
5. W. Luk, S. Guo, N. Shirazi and N. Zhuang, "A framework for developing parametrised FPGA libraries", in *Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, LNCS 1142, Springer, 1996.
6. W. Luk, N. Shirazi and P.Y.K. Cheung, "Modelling and optimising run-time reconfigurable systems", in *Proc. FCCM96*, IEEE Computer Society Press, 1996.
7. W. Luk, N. Shirazi and P.Y.K. Cheung, "Compilation tools for run-time reconfigurable designs", in *Proc. FCCM97*, IEEE Computer Society Press, 1997.
8. P. Lysaght, "Towards an expert system for a priori estimation of reconfiguration latency in dynamically reconfigurable logic", in *Field Programmable Logic and Applications*, LNCS 1304, Springer, 1997.
9. N. Shirazi, W. Luk and P.Y.K. Cheung, "Automating production of run-time reconfigurable designs", in *Proc. FCCM98*, IEEE Computer Society Press, 1998.
10. M.J. Wirthlin and B.L. Hutchings, "A dynamic instruction set computer", in *Proc. FCCM95*, IEEE Computer Society Press, 1995.